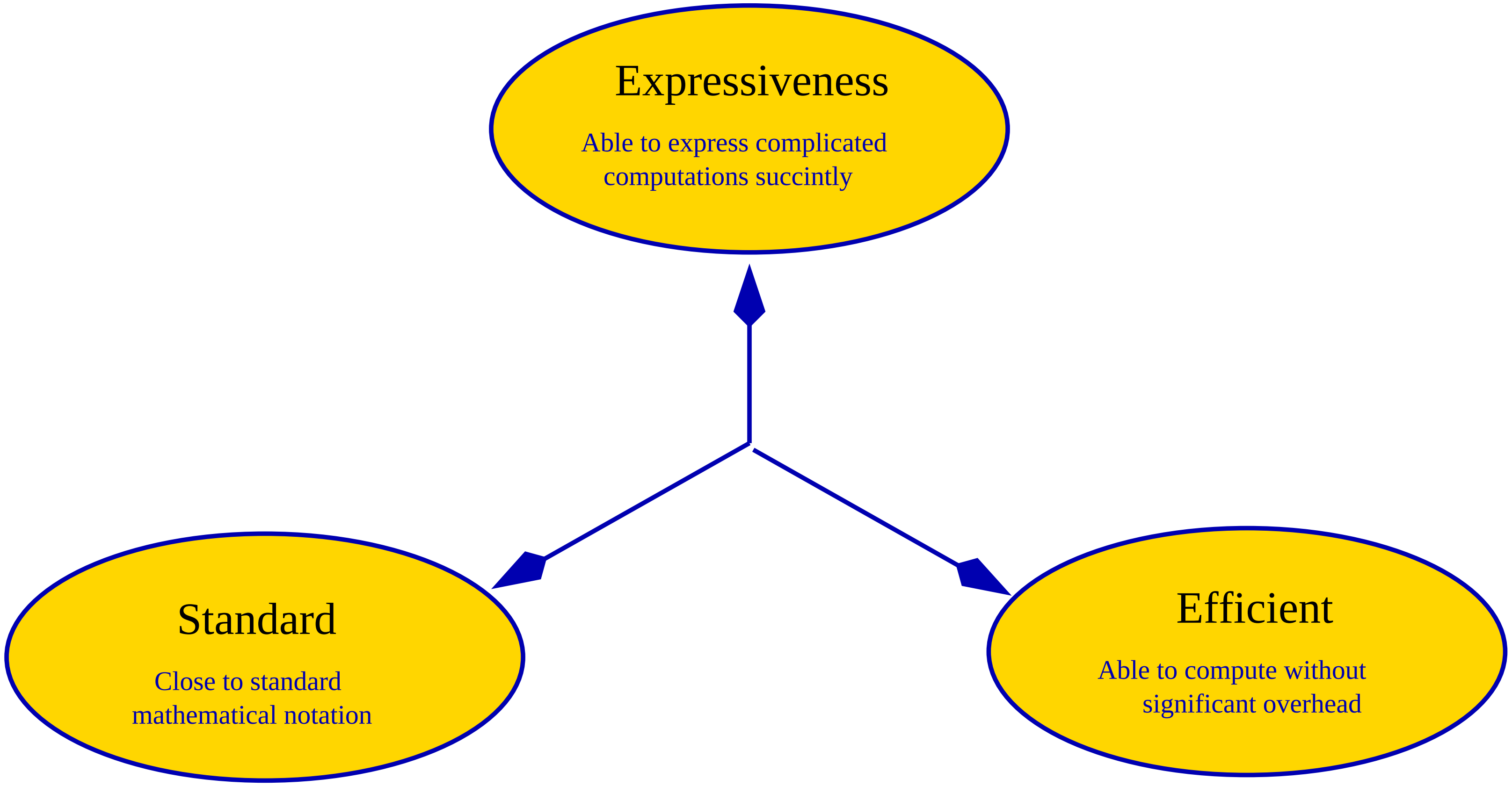
Numerical Analysis, Computer Algebra and Bioinformatics: Having fun with optimization

Gaston H. Gonnet
ETH Informatik
Zürich, Switzerland

SVG, Stanford, January $9-10^{\text{th}}$, 2004

The substance

- Language design for Linear Algebra has a long way to go
- Computing $A^t A$ efficiently
- Computing some vector operations efficiently



Good Design: a delicate balance between three objectives

Linear Algebra in Darwin

- Lists, [1,2,3...] are vectors, matrices are lists of lists
- Automatic arithmetic (incl powers) on vectors and matrices
- Darwin is object oriented
- Functions extended to work on lists: max, min, sum
- Special functions over matrices, in particular exp, ln
- zip and seq, option zippable

Computing A^tA efficiently

This is a very common computation, it is usually required for various forms of regression computation. Typically in the context of trying to find a vector x which approximates $Ax \approx b$. In this context we also want to compute A^tb and b^tb.

- Offline vs Online methods (or incremental computation)
- Five (2+3) computation methods in Darwin
- Radically different complexities

Offline methods

Assuming that all the matrix A and vector b are available at once.

Direct method

```
AtA1 := A^t * A:
btA := b*A:
btb := b*b:
```

Use internal products, compute only half of the matrix

```
At := A^t:
AtA2 := CreateArray(1..m,1..m):
for i to m do for j from i to m do
    AtA2[i,j] := AtA2[j,i] := At[i] * At[j] od od:
btA := b*A:
btb := b*b:
```

Online methods

Here we assume that the rows of A are given one at a time (together with the corresponding element of b).

Using element-times-row

```
AtA3 := CreateArray(1..m,1..m):
btA := CreateArray(1..m):  btb := 0:
for i to n do
    Ai := A[i];
    for j to m do AtA3[j] := AtA3[j] + Ai[j]*Ai od;
    btA := btA + b[i]*Ai;
    btb := btb + b[i]^2
    od:
```

Explicit loops

```
AtA4 := CreateArray(1..m,1..m):
btA := CreateArray(1..m):
btb := 0:
for i to n do
    Ai := A[i];
    for j to m do
        for k from j to m do
            AtA4[j,k] := AtA4[j,k] + Ai[j]*Ai[k] od od;
    btA := btA + b[i]*Ai;
    btb := btb + b[i]^2
    od:
for j to m do for k from j+1 to m do AtA4[k,j] := AtA3[j,k] od od;
```

Using exterior products

```
AtA5 := CreateArray(1..m,1..m):
btA := CreateArray(1..m):
btb := 0:
for i to n do
    Ai := A[i];
    AtA5 := AtA5 + [Ai]^t * [Ai];
    btA := btA + b[i]*Ai;
    btb := btb + b[i]^2
    od:
```

Which is faster?

Timings for a random matrix A(5000,20) run on an AMD 2.2.

method	seconds
Direct	0.17
Internal products	0.05
Element-times-row	0.37
Explicit loops	1.40
Exterior products	0.24

Computing some vector operations efficiently

We coded 2 very common array operations `A = A+c*b` and `A*B` in 9 different forms (as macros). These macros are used throughout the kernel code.

A pre-processing phase (during system building time) decides automatically which is the most efficient implementation and uses it.

Macros

The macro VectorAcB(A,c,B,fr,to) adds the vector c*B to the vecor A. This is the operation responsible for the $O(n^3)$ behaviour in Gaussian elimination and many other cubic matrix algorithms. In its simplest form it is coded as:

```
for( i=fr; i<to; i++ ) A[i] += c * B[i];
```

The internal product macro, VectorAB(A,B,fr,to,r) is:

```
for( r=0, i=fr; i<to; i++ ) r += A[i] * B[i];
```

Other forms of the macro VectorAcB

Using pointer arithmetic

```
{ double *vecA={A+(fr)}, scalarc={c}; \
    int vc={(to)-(fr)}, BmA={(B)-(A)}; \
    while( vc-- > 0 ) \
        { vecA[0] += scalarc * vecA[BmA];  vecA++; }}
```

Using two independent pointers

```
{ double *vecA={A+(fr)}, *vecB={B+(fr)}, scalarc={c}; \
    int vc={(to)-(fr)}; \
    for( ; vc>0; vc-- ) \
        { vecA[0] += scalarc * vecB[0];  vecA++;  vecB++; }}
```

Big surprises

- VectorAcB is 6 times faster than VectorAB (for AMD, 2 for Intel)
- `for( ; vc>0; vc-- )` very different from `while( vc-- > 0 )`
- Optimization level does not affect icc results
- Quite often higher optimization is slower for gcc
- Code duplication and triplication are worse

- Most C programmers believe that all versions would be equiv-alent

- No
  - compiler or
  - optimization level or
  - flag settings or
  - version of the macros

  is uniformly better

Why? What to do?

The reasons for these substantial differences are difficult to explain. We conjecture a mixture of:

- Peculiar architecture of the Pentium instruction set

- Pipelining

- Desire to score well in some standard tests

- Bugs

Who cares? Just select the right one (faster one) and leave this problem to compiler buffs.